Efficient Identification of Regular Expressions from Representative Examples

Alvis Brāzma
Institute of Mathematics and Computer Science
University of Latvia[1]
29 Rainis blvd., Riga, LV-1459, Latvia
E-mail: abrazma@mii.lu.lv

Abstract

Learning of regular languages representable as regular expressions without a union operation is considered. Rough equivalence of regular expressions is defined in the way, that it does not distinguish between expressions of the type $(A)^*$ and $(A^n)^*$ or $(A)^*A^n$. A notion of c-uniform example of a regular expression is introduced. An example is c-uniform if all the iterations are "unfolded" about the same number of times. It is proved that any regular expression which does not contain union operations can be approximately identified up to the rough equivalence from a long enough c-uniform example in polynomial run-time. In other words, for any regular expression E not containing a union operation, given a long enough c-uniform example of E, a regular expression F such that F is roughly equivalent to E can be found in polynomial run-time. The identification algorithm is generalized to the effect, that it works for a much wider class of examples. A computer experiment of learning regular expressions via the proposed algorithms is reported, and encouraging results suggesting that the algorithms are quite practical are presented.

1 Introduction

Historically the first model of computational learning was introduced by E.M.Gold [10]. Let $\mathcal{L}$ be a language over some alphabet Σ, and let S be an infinite sequence $(l_1, s_1), \ldots, (l_k, s_k), \ldots$, where $l_i \in \Sigma^*, t_i \in \{0, 1\}$, and $t_i = 1$ if $l_i \in \mathcal{L}$; $t_i = 0$ if $l_i \notin \mathcal{L}$. An algorithm $\mathcal{I}$ is said to *identify a language $\mathcal{L}$ in representation $\mathcal{R}$ in the limit* from sequence S, if there exists k such that after receiving the first k examples of S the algorithm outputs a correct description of the language $\mathcal{L}$ in terms of $\mathcal{R}$. The algorithm $\mathcal{I}$ is said to identify $\mathcal{L}$ in polynomial time if the updating time for each next hypothesis is at most some polynomial in the size of $\mathcal{L}$ and the sum of lengths of data provided. For more formal definitions see [12].

It is not a coincidence that one of the classes of languages the learning of which is studied most is regular languages. Regular languages is one of the fundamental notions of computer science which has the most extensive practical applications. Even a partial solution to the problem of the learning of regular languages would have a considerable practical importance. For instance, it may help to approach the problem of program synthesis from examples.

From the results of Gold [10] it follows that the class of regular languages is identifiable in the limit and is not identifiable in the limit from positive examples (i.e., such that $t_i = 1$) only. In [1] Angluin showed that the whole class of regular languages cannot be polynomially identified in the limit in the representation of deterministic finite state automata (DFA). This and some other negative results on efficient learning of regular languages have encouraged researchers to look for new approaches. Among them are extending the model of learning in the way that more information is available to the learner (e.g., introducing queries [2, 3]) and restricting the class of languages which has to be learned by a particular method [8, 13]. Other possibilities are trying to learn the language only approximately (e.g., PAC-learning [14]) or learning from specific kind of "good" examples [9, 11, 15].

In [13] a class of the so called strict DFAs is defined and proved to be polynomially identifiable from positive examples. Another restricted class of DFA is studied in [8]. At the same time very little is known about polynomial learning of regular languages represented as regular expressions, especially without powerful oracles. The learning of regular languages in the representation of regular expressions is a very important and interesting problem since a regular expression is a very direct and

ACM COLT '93 /7/93/CA, USA
© 1993 ACM 0-89791-611-5/93/0007/0236. .$1.50

[1]The reported research was partially done while the author was in New Mexico State University

compact representation of the language. Also, a regular expression is a very natural way to describe different kinds of periodic repetitions such as computational traces generated by while loops.

In [7] a class of regular expressions which can be polynomially identified from one "good" example is defined. In this paper we show that any regular expression can be reasonably approximated in polynomial time given an appropriate set of representative examples. The paper consists of two parts. In the first part the notions of c-uniform example and rough equivalence are defined and it is proved that any regular expression can be identified in polynomial time up to the rough equivalence from an appropriate set of long enough c-uniform examples.

The second part of the paper describes a natural generalization of the algorithm studied in the first part and presents encouraging experimental results of learning via the generalized algorithm. This represents a quite different approach to the problem of learning: to take a natural learning algorithm itself as a basis for research and to study what classes of languages can be learned by it. Although some may argue that the meaning of "natural" is ambiguous, to our belief this approach may turn out to be more practical as purely theoretical approach. Quite often it is difficult to formulate the precise boundaries of applicability of practically working algorithms. Note that similar approach is followed in [4] for learning arithmetic formulas.

2 Theoretical Approach

Let Σ be a finite alphabet. Regular expressions are defined inductively: any $E \in \Sigma^*$ is a regular expression; if E and F are regular expressions, then concatenation: EF, union: $E \cup F$, and iteration: $(E)^*$ are also regular expressions. The language $\mathcal{L}(E)$ generated by the regular expression E is defined as follows: if $E \in \Sigma^*$, then $E = \mathcal{L}(E)$; $\mathcal{L}(EF) = \{XY | X \in \mathcal{L}(E), Y \in \mathcal{L}(F)\}$, $\mathcal{L}(E \cup F) = \mathcal{L}(E) \cup \mathcal{L}(F)$, and $\mathcal{L}((E)^*) = \mathcal{L}(E^n)$, $n \geq 0$ (E^0 is the empty expression). E and F are called *equivalent* ($E \equiv F$) if $\mathcal{L}(E) = \mathcal{L}(F)$. For brevity, instead of $X \in \mathcal{L}(E)$ we will simply write $X \in E$.

Let us say that a regular expression E is in normal form if $E = E_1 \cup \ldots \cup E_k$, where $E_1, \ldots, E_k$ are regular expressions containing iteration and concatenation operations only. We call such E_i a *$*$-expression*. Note that for any regular expression there exists an equivalent regular expression in a normal form (since $(F_1 \cup \ldots \cup F_k)^*$ is equivalent to $(F_1^* \ldots F_k^*)^*$). Certainly, there may be many normal forms for the same expression.

A *term* is a $*$-expression of the type $(T)^*$, where T is an arbitrary $*$-expression. An *unfolding* of a term $(T)^*$ is any expression $TT \ldots T = T^n$, for $n \geq 0$. n is called the *factor* of the unfolding of term $(T)^*$. Below, instead of $(T)^*$ we will frequently write simply T^*. Let us define an *unfolding* of a $*$-expression E inductively as follows:

1. The expression E itself is an unfolding of E;
2. If an expression $E_1 T^* E_2$ is an unfolding of E, then, for any $n \geq 0$, the expression $E_1 T^n E_2$ is an unfolding of E.

For instance, if the $*$-expression is $a(bc^*)^*d^*$, then abc^*d^*, $abc^*bc^*bc^*bc^*d^*$, $abc^*bc^*bccccbbc^*dddd$, $abcccbccbccccbbccccdddd$ are some of the unfoldings.

If an unfolding X of a $*$-expression E is such that it does not contain any terms, we will call it a *complete unfolding* or an *example* of the $*$-expression. Note, that any $X \in E$ can be obtained as a sequence of expressions $E_0, E_1, \ldots, E_n$, where $E_0 = E$, $X = E_n$ and E_{i+1} is obtained from E_i by replacing a term in E_i by its unfolding. $E_0, E_1, \ldots, E_n$ is called an *unfolding sequence leading from E to X*.

Let f_{max} (f_{min}) be the maximal (minimal) factor of any unfolding of a term applied in a given unfolding sequence S. We call the sequence S *c-uniform* ($c \geq 1$) if $f_{max}/f_{min} \leq c$. We call the unfolding F of E *c-uniform* if there exists a c-uniform unfolding sequence leading from E to F. We call the unfolding *expressive* if $f_{min} \geq 1$. For instance, for the unfolding: $abcccbccbccccbcccbcc ccdddd : f_{min} = 2, f_{max} = 5$. Informally the fact that an example is c-uniform means that to obtain it all the iterations of the expression are unfolded more or less the same number of times. To our mind this is a pretty reasonable assumption for examples which can be considered "good", although it has some drawbacks (see Section 4).

In [7] a subclass of $*$-expressions which can be identified by one long c-uniform example is defined in the following way. Let us call a $*$-expression *unambiguous* if the following two properties hold:

1. E does not contain any term of the type $((T)^*)^*$ or $(T^n)^*$ for $n > 1$;
2. If there exists an expressive unfolding $F = F_1 G_1 (T)^* G_2 F_2$ of the expression E such that $f_{min} \geq 1$ and $G_2 \equiv T$ or $T = T_1 T_2$ and $T_2 \equiv G_1$, then there exists another expressive unfolding $F' = F_1 H_1 (T)^* H_2 F_2$ such that $H_1 \supset F_1$, $H_2 \supset F_2$, $H_1 \not\equiv F_1$ and $H_2 \not\equiv F_2$.

The simplest instances of expressions which are not unambiguous are $(aa)^*$ (because of 1.), $a^*a, aa^*, a(ba)^*$ (all, because of 2.). Also note that $ab^*(cb^*)^*d$ is not unambiguous ($G_1 = b^*, T_2 = b^*$), but $ab^*(cb)^*d$ is unambiguous, since, even if for $G_1 = b, T_2 = b$ the premise of 2. holds, there exists $H_2 = b^*$ such that $H_2 \supset G_2$. Similarly, $a(cb^*)^*(cb^*d)^*$ is not unambiguous ($F = a(cb^*)^*cb^*d$, $G_2 = cb^*$), but $a(cb^*)(c^*b^*d)^*$ is unambiguous.

It is proved in [7] that the following theorem holds:

Theorem 1 *There exists an algorithm $\mathcal{L}$ such that for any unambiguous expression E and any constant $c \geq 1$ there exists an integer $l \in \mathbf{N}$ such that, given a c-uniform example $X \in E$ of the length greater or equal to l (i.e., $|X| \geq l$), the algorithm $\mathcal{L}$ outputs the expression*

E in run-time $O(|X|^3)$.

In this paper we use a different approach: we do not restrict the class of expressions, instead, we try to identify the expression only approximately to within a certain degree of accuracy.

Let us say that a $*$-expression F can be obtained from $*$-expression E by a *loop preserving transformation*, if F can be obtained from E by replacing a substring in any of the following ways:

1. $((A)^*)^* \leftrightarrow (A)^*$;
2. $(A)^*(A)^* \leftrightarrow (A)^*$;
3. $A(BA)^* \leftrightarrow (AB)^*A$;
4. $(A)^*A \leftrightarrow (A)^*$;
5. $(A^n)^* \leftrightarrow (A)^*$,

where A and B are arbitrary $*$-expressions and $n \in \mathbf{N}^+$. Note that the transformations 1-3 preserve equivalence, but 4 and 5 do not.

We say that $*$-expressions E and F are *roughly equivalent*, if there exists a sequence of $*$-expressions $E_0, \ldots, E_n$, such that $E = E_0$, $F = E_n$, and E_{i+1} is obtained from E_i by a loop preserving transformation. For instance, $(a^*ba^*)^*$ and $(a^*b)^*a^*$ are roughly equivalent, since the transformations: $(a^*ba^*)^* \rightarrow (a^*ba^*)^*a^*ba^* \rightarrow a^*(ba^*a^*)^*ba^* \rightarrow a^*(ba^*)^*ba^* \rightarrow (a^*b)^*a^*ba^* \rightarrow (a^*b)^*a^*$ are possible.

Note that the rough equivalence disregards the "counting" properties of the expressions but preserves the "loop structure": no iteration can be lost or created as the result of the transformations. It may be argued that this approach (i.e., learning up to rough equivalence) deliberately avoids the most difficult problem in learning regular expressions: "countingness". This is clearly so, our justification for the approach is that in practical applications of regular expressions the "countingness" very often is not the most important aspect and it may be enough to restore just the loop structure of the expression.

Theorem 2 *There exists an algorithm $\mathcal{L}$ such that for any $*$-expression E and any constant $c \geq 1$ there exists an integer l such that, given a c-uniform example $X \in E$ of the length greater or equal to l (i.e., $|X| \geq l$), the algorithm $\mathcal{L}$ outputs an expression F roughly equivalent to E in run-time $O(|X|^3)$.*

Arbitrary regular expressions E and F are said to be *roughly equivalent*, if there exist normal forms $E_1 \cup \ldots \cup E_k$ and $F_1 \cup \ldots \cup F_k$ of E and F respectively, such that for any $i = 1, \ldots, k$, E_i is roughly equivalent to F_i. From Theorem 2 we immediately get:

Corollary 1 *There exists and algorithm $\mathcal{L}'$ such that for any regular expression in normal form $E = E_1 \cup \cdots \cup E_k$ and any constant $c \geq 1$ there exists an integer l such that, given c-uniform examples $X_1 \in E_1, \ldots, X_k \in E_k$ such that $|X_1| \geq l, \ldots, |X_k| \geq l$, the algorithm $\mathcal{L}$ outputs an expression F roughly equivalent to E in run time $O(|X_1| + \cdots + |X_k|)^3$.*

3 The Algorithm

In this section we will define the algorithm and sketch the proof of Theorem 2.

Let us call a $*$-expression E *periodic with the factor f ($f \in \mathbf{N}^+$) and the period p ($p \in \mathbf{N}^+$)*, if there exists an expression T such that $|T| = p$ and $E = T^f$. T is called the *body* of the periodic expression E. For instance, expression $ab^*ab^*ab^*ab^*ab^*$ is periodic with the period 3, factor 5 and ab^* is the body. It is evident that the following lemma holds:

Lemma 1 *Given an expression E and an integer $p > 0$, an integer f can be found in time $O(|E| + p)$ such that E has a prefix which is a periodic expression with the period p and factor f and does not have a prefix which is a periodic expression with the period p and factor $f' > f$.*

The algorithm $\mathcal{L}$ of the theorem is the following. Given an example X, first it finds $f_0 = \log(|X|)$, and then runs the procedure **SYNTHESIS** in figure 1. The idea of the procedure is that for all possible periods $p = 1, 2, \ldots$ it checks whether the substring beginning with $X[0], X[1], \ldots$ is a periodic subexpression and finds the maximal appropriate factor f. If f is at least f_0, then the procedure replaces the substring T^f, where T is the body of the periodic expression, by the term $(T)^*$. For instance, given the example *abababbbabababbbabababbbabababbba* of the expression $((ab)^*b)^*a$, the algorithm will first construct $(ab)^*(ba)^*b(ba)^*b(ba)^*bba$, and then $(ab)^*((ba)^*b)^*ba$ which is roughly equivalent to the original expression. Note that this method is very similar to the one proposed in [6].

Let us also note that, although the worst case performance of the algorithm is $O(|X|^3)$, it can occur only if the length of the expression E is very close to the length of the example X. If, for given E, $|X| \rightarrow \infty$, then the performance of the algorithm is only $o(|X|^2)$.

Now let us sketch the proof that the described algorithm satisfies Theorem 2. In fact we prove a more general theorem claiming that any $*$-expression E can be effectively transformed into, the so called, standard form, which is roughly equivalent to E such that, given a long enough c-uniform example X of E, the algorithm $\mathcal{L}$ outputs the standard form of E (Theorem 3). Here the basic idea is to prove that, if X is a sufficiently long c-uniform example of a $*$-expression E, then for any $n = 1, 2, \ldots$, the expression E can be effectively transformed into E_n roughly equivalent to E, and there exists an unfolding F_n of E_n such that the expression obtained after n executions of the replacing step in the procedure **SYNTHESIS** coincides with F_n. Theorem 3 (and 2) will follow by in-

```
procedure SYNTHESIS(X, f_0) return expression;
    for p = 1 to |X|/f_0 do;
        j := 0;
        while j ≤ |X| − p do;
            if X[j..(j + p − 1)] is a *-expression then do;
                find the maximal integer f such that
                X[j..(j + f * p − 1)] is a periodic expression
                with the period p;
                if f ≥ f_0 then do;
                    replace X[j..(j + f * p − 1)] in X by
                    (X[j..(j + p − 1)])*;
                    j := j + 3;
                end;
                else j := j + (f − 1) * p − 1;
            end;
            else j := j + 1;
        end;
    end;
    return X;
end procedure;
```

Figure 1: Procedure **SYNTHESIS**

duction. To follow the sketched thread, several lemmas are needed.

Lemma 2 *Let X and Y be two periodic expressions with the bodies T_x and T_y, respectively. Let T_x and T_y be such that there does not exist T such that $T_x = (T)^m$ or $T_y = (T)^m$, for $m > 1$, neither for T_x nor T_y. Further, let X and Y have a common substring of the length at least $|T_x| + |T_y|$. Then there exist T_1 and T_2 such that $T_x = T_1T_2$, $T_y = T_2T_1$ (T_2 may be empty, then $T_x = T_y$).*

The proof of this lemma is straight forward.

Let ϕ be a polynomial. Let us call an unfolding F of a $*$-expression E *ϕ-uniform*, if there exists an unfolding sequence leading from E to F such that for the minimal and maximal factors f_{min} and f_{max} in F: $f_{max} \leq \phi(f_{min})$. Thus, taking $\phi(x) = cx$ we get c-uniformity. Note that for any expression E and polynomial ϕ there exists an $l > 0$ such that, if X is a ϕ-uniform unfolding of E and $|X| \geq l$, then $f_{min} > \log|X|$ and X contains a periodic subexpression with the factor at least f_{min}.

Now, let us consider a $*$-expression E. We say that a *term T^* can be shifted leftwards by a substring* T_2 in expression E, if $T = T_1T_2$ for some T_1 and either T^* is in a substring T_2T^*, or in a substring $(T^*F)^*$ for some F and the term $(T^*F)^*$ can be shifted leftwards by T_2. The term $(T^*F)^*$ is called the *nesting* term for T^*. By *shifting* the term $(T_1T_2)^*$ in E we will understand the following transformation: if the term is in a substring $T_2(T_1T_2)^*$, then we replace the substring by $(T_2T_1)^*T_2$; if the term is in a substring $((T_1T_2)^*F)^*$, then we first shift the nesting term by T_2 and then we shift $(T_1T_2)^*$. In the second case we say that the term $(T_1T_2)^*$ *induces shifting* of the nesting term. For instance, $(ab)^*$ can be shifted leftwards by b in $bb((ab)^*bb)^*$, and the result of shifting is $b((ba)^*bb)^*b$.

A term T^* *can be expanded rightwards* if it is in a substring T^*T or T^*T^*. By *expanding* we understand substituting T^* for any of these substrings. We say that a term T^* can be simplified if $T = (T')^*$. We will use the procedure of simplification replacing the terms of the type $(T^*)^*$ by T^*. We say that the term T^* of E is the *left maximal*, if it cannot be shifted leftwards, nor expanded, nor simplified.

Finally, let the algorithm of standardization transform an arbitrary $*$-expression into the so called *standard form* in the following way. The algorithm first replaces all the terms T^* in the expression E by TTT^*TT. Then, beginning with the terms with the smallest periods and from the left side, it maximally shifts them to the left, expands and simplifies. While shifting a term a shifting of the nesting terms may be induced. Note, that standardization can be accomplished by loop preserving transformations and therefore preserve the rough equivalence.

Let us denote by $\mathbf{STANDARD}_p(E)$, $p \in \mathbf{N}$ the expression obtained from the expression E by applying the standardization algorithm up to the point when all the terms T^* such that $|T| \leq p$ are shifted leftwards and expanded. Let us call an unfolding F of a $*$-expression E *right-to-left-top-down* or simply *r-l-t-d* if the unfolding sequence $E_0, \ldots, E_n$ leading from E to F is such that for any i, while unfolding E_i, always the rightmost term with the maximal period is unfolded. We say that the *level* of the given *r-l-t-d* unfolding of E is p ($p \in \mathbf{N}$), if all the terms with the periods greater than p and no term with the period smaller than p are unfolded. It can be proved

that, if F is a p-level r-l-t-d-unfolding of an expression $\mathbf{STANDARD}_p(E)$, then all terms T^* of F such that $|T| \leq p$ are the left maximal and $\mathbf{STANDARD}_p(F) = F$.

Let XYZ be an expression and Y be a periodic subexpression with the body U. We call Y *locally maximal* with the given period, if U is neither prefix of Z, nor suffix of X. We call Y *locally leftmost* if there do not exist U_1 and U_2 such that $U = U_1U_2$ and U_2 is a suffix of X. Let us note that, if an expression $E_1(T)^*E_2$ is in the standard form, then the periodic subexpression T^n in $E_1T^nE_2$ is locally maximal and leftmost.

Lemma 3 *For any expression E, polynomial ϕ, and integer p, there exists an integer $f_0 > 0$, such that, if $Y = Y_1XY_2$ is a ϕ-uniform p-level r-l-t-d-unfolding of $\mathbf{STANDARD}_p(E)$ and X is the first from the left locally maximal and leftmost periodic subexpression with the body Z, period p, and factor $f \geq f_0$, then the expression $Y' = Y_1(Z)^*Y_2$ is also a ϕ-uniform p-level r-l-t-d-unfolding of $\mathbf{STANDARD}_p(E)$.*

The sketch of proof. Since the unfolding is ϕ-uniform, if the factor f in the lemma is large enough, X should intersect with an unfolding H of a certain term $(T)^*$ of E on an arbitrary long substring. Taking into account that $|Z| \leq |E|$, it follows from Lemma 2 that T coincides with Z "to within shift" (i.e., there exist Z_1 and Z_2 such that $Z = Z_1Z_2$ and $T = Z_2Z_1$), but since E is in the standard form on the one hand and X is locally maximal and leftmost on the other hand, we get that T and Z coincide completely: $T = Z$. Moreover, $H = X$. Now, if we unfold the expression in the same way as if to obtain Y, except for that we do not unfold the term $(T)^*$ to obtain X, then we get the required unfolding $Y_1(Z)^*Y_2$, from where follows the lemma.

Let us denote by $\mathbf{SYNTHESIS}_p^i(X)$ the expression obtained during the running of the algorithm $\mathcal{L}$ just after procedure SYNTHESIS has completed the replacement of the first i periodic substrings with the periods less or equal to p. Then, using lemma 3, the following lemma, establishing relation between $\mathbf{STANDARD}_p(E)$ and $\mathbf{SYNTHESIS}_p^i(X)$, can be proved by induction.

Lemma 4 *For any $*$-expression E and polynomial ϕ, there exist an integer l and polynomial ϕ' such that for any ϕ-uniform example $X \in E$ such that $|X| \geq l$, and for any integer $p \leq |E|$ and i, there exists a ϕ'-uniform r-l-t-d-unfolding F^i of the expression $\mathbf{STANDARD}_p(E)$ such that $F^i = \mathbf{SYNTHESIS}_p^i(X)$.*

Now, it follows by induction from lemma 4 that the following theorem holds:

Theorem 3 *For any $*$-expression E and any polynomial ϕ, there exists an integer l such that, given a ϕ-uniform example $X \in E$ of length at least l, the algorithm $\mathcal{L}$ outputs an expression F such that F is the standard form of E in run-time $O(|X|^3)$.*

Taking $\phi(x) = cx$ we can obtain Theorem 2 as a corollary. Theorem 3 is in fact stronger than Theorem 2, also because it states that algorithm $\mathcal{L}$ restores the expression in a particular form, which intuitively may be considered as the simplest regular expression having the same loop structure as the original.

4 Generalized Algorithm and Computer Experiments

The learning algorithm $\mathcal{L}$ has a clear bottle-neck: even small "irregularities" in the examples may tamper with the learning. For instance, the algorithm will not be able to learn expression $(ab^*)^*$ from example *abbbbbbabbbbbabbbbbbbabaabbbbbbabbbbbb* because of just one irregularity: just once b^* is unfolded only one time. On the other hand, for a human it would be very natural to recognize the underlying expression.

Therefore it is important that it is possible to generalize the learning algorithm $\mathcal{L}$ to the effect that it can learn from a much broader class of examples including the one above. The generalized version of the algorithm should recognize the subexpressions like $ab^*ab^*ab^*abab^*ab^*$ as kind of periodic and be able to substitute it by term $(ab^*)^*$. It means that the notion of "periodicity" used in the generalized algorithm cannot be entirely syntactic.

To deal with this more complicated notion of periodicity, the generalized algorithm, which we will denote by $\mathcal{L}_1$, remembers during the operation not only the expression generated so far, but also the respective parts of the example, each part of the generated expression comes from. For this we need the notion of *annotated expression* $\overline{E}$ which is defined as a pair (E, S), where E is a $*$-expression and S is, the so called, *unfolding scenario*. Intuitively, the scenario describes how to unfold the expression to obtain a particular example. Formally, an unfolding scenario is defined inductively:

- An empty word is an unfolding scenario;
- If S_1 and S_2 are scenarios, then S_1S_2 is a scenario;
- If $S_1, \ldots, S_n$ are scenarios, then $n(S_1 \ldots S_n)$ is a scenario.

The *unfolding* $Unf(E, S)$ *of the expression* E *according to the scenario* S is defined as follows:

- If S is empty, then $Unf(E, S) = E$;
- If S is not empty, but E does not contain any term (i.e., the star-height of E is 0), then $Unf(E, S)$ is undefined;
- If $E = FT^*E'$, where T^* is the leftmost term in E (i.e., the star-height of F is 0), then $Unf(FT^*E', n(S_1 \ldots S_n)S') = Unf(FT^nE', S_1 \ldots S_nS')$.

For instance $Unf((ab^*)^*cd^*, 3(1(\,)2(\,)3(\,))4(\,)) = aba\,bbabbbcdddd$. Note that $3(1(\,)2(\,)3(\,))4(\,)$ is actually a parse tree for *ababbabbbcdddd*.

Below, writing $\overline{E} = (E, S)$ we will assume that $Unf(E, S)$ is defined. If $Unf(E, S)$ is defined and $E = E_1E_2$, then S can be uniquely represented in the form $S = S_1S_2$ such that $Unf(E, S) = Unf(E_1, S_1)Unf(E_2, S_2)$. We will say that S_i *corresponds to* E_i, $i = 1, 2$, and write $\overline{E} = \overline{E_1}\,\overline{E_2}$, $\overline{E_1} = (E_1, S_1)$, $\overline{E_2} = (E_2, S_2)$.

Given a $*$-expression E and a word X, the algorithm $\mathcal{L}_1$ will need to find a scenario S such that $Unf(E, S) = X$, if $X \in E$. Unfortunately in the general case this problem is $\mathcal{NP}$-complete, since it involves determining whether $X \in E$ and finding a parse tree for a regular expression E. The way we approach the problem is that we define a polynomial-time algorithm about which we can prove that for some particular subclass of expressions, which we call *totally unambiguous*, the algorithm works always correctly. If E does not belong to the class, the procedure may also fail to find S, even if in fact $X \in E$, but it will not spend more than polynomial in $|X|+|E|$ time. Nevertheless, as it is supported by experimental results reported below, actually, the procedure works correctly for a much wider class of expressions, unless the example X is "unfortunate". (Therefore, we do not present the definition of totally unambiguous expressions.)

In fact, algorithm $\mathcal{L}_1$ will need more general algorithms **PREFIX**(E, X) and **SUFFIX**(E, X), which determine whether there exists a prefix (suffix) Y of X, such that $Y \in E$, and returns the respective unfolding scenario.

The algorithm **PREFIX**(E, X) uses a greedy strategy. Let $E = X_1(T)^*E_1$. If there is a prefix Y such that $Y \in E$, then $X = X_1X_2$, and the algorithm calls a procedure **COVERRIGHT**(T, X_2) which tries to find the maximal integer n such that X_2 has a prefix $Y_2 \in T^n$. Procedure **COVERRIGHT**(T, X_2), in turn, repeatedly recursively calls **PREFIX**(T, X_2), cutting off the prefix from X_2 in the case of success (note, that the star-height of T is less than that of E, therefore the recursion will halt). No backtracking is used: after the first failure of **PREFIX** procedure **COVERRIGHT** halts. Next **PREFIX** recursively calls itself on E_1 and the respective suffix of X. Parallelly n is used to construct the scenario.

Similarly, procedures **SUFFIX** and **COVERLEFT** can be defined. In fact, we need to generalize the procedures **COVERRIGHT** and **COVERLEFT** to the effect that they can be applied to an expression E and an annotated expression $\overline{F} = (F, S_F)$. **COVERRIGHT**$(E, \overline{F})$ (**COVERLEFT**$(E, \overline{F})$) tries to cover $Unf(F, S_F)$ from the left (right) side with E^n for n as large as possible. It returns a scenario S_{E^n}, called the *covering scenario*, an annotated expression $\overline{F'} = (F', S_{F'})$, called the *remaining expression*, and an integer n, called the *covering factor*, such that $Unf(E^n, S_{E^n})Unf(F', S_{F'}) = Unf(F, S_F)$ $(Unf(F', S_{F'})Unf(E^n, S_{E^n}) = Unf(F, S_F))$. The procedure works by gradually unfolding F from the left (right) side according to the scenario S_F.

Now, let us define the notion of a *semiperiodic* subexpression with the given *body* and *factor*. Let $\overline{E} = (E, S) = (FTG,\ S_FS_TS_G)$ be an annotated expression. We say that $\overline{E}$ contains a *semiperiodic* subexpression with the given body T and factor f, if there exist words X and Y and integers f_X, f_Y such that $Unf(F, S_F) \in XT^{f_X}$, $Unf(G, S_G) \in T^{f_Y}Y$ and $f_X + f_Y + 1 = f$.

Given an annotated expression $\overline{E}$ and a subexpression T of E, procedures **COVERRIGHT** and **COVERLEFT** can be used to find f such that an annotated expression $\overline{E} = (\overline{F}\,\overline{T}\,\overline{G}) = (FTG, S_FS_TS_G)$ is semiperiodic with the body T and factor f. Namely, let procedure **COVERLEFT**$(T, \overline{F})$ return the remaining expression $\overline{F'} = (F', S_{F'})$, the covering scenario S_{TR} and the covering factor f_X. Let procedure **COVERRIGHT**$(T, \overline{G})$ return the remaining expression $\overline{G'} = (G', S_{G'})$, the covering scenario S_{TL} and the covering factor f_Y. Then $\overline{E}$ contains a semiperiodic substring with the body T and the factor $f = f_X + f_Y + 1$. It can be proved that, if the expression E is totally unambiguous, then f is the maximal possible. Let us define the operation **CREATETERM** as substituting the annotated expression:

$$\overline{E'} = (F'(T)^*G',\ S_{F'}f(S_{TL}S_TS_{TR})S_{G'}),$$

for the expression:

$$\overline{E} = (FTG,\ S_FS_TS_G).$$

Now algorithm $\mathcal{L}_1$ can be described almost in the same way as algorithm $\mathcal{L}$, except that instead of looking for the maximal integer f, such that there is a periodic substring with the chosen body T and the factor f, algorithm $\mathcal{L}_1$ uses procedures **COVERRIGHT** and **COVERLEFT** for finding the integer f such that there is a semiperiodic substring with the body T and the factor f. If $f \geq f_0$, then, instead of just replacing the regular substring by the appropriate term as in $\mathcal{L}$, the algorithm $\mathcal{L}_1$ uses the above described operation **CREATETERM** adjusting also the scenario. Thus, beginning with an example X and the empty scenario, the algorithm $\mathcal{L}_1$ concludes with some expression F, which we will call the *synthesized expression*, and scenario S such that $Unf(F, S) = X$.

The algorithm has polynomial run-time for any $*$-expression E and any example X. It can be proved that, if the expression E is totally nonambiguous and the example X is c-uniform and long enough, then the synthesized expression coincides with E. We do not prove this assertion here since it is weaker than the results for the algorithm $\mathcal{L}$. On the other hand, intuitively it seems that the algorithm $\mathcal{L}_1$ should be statistically better. To study the statistical behavior of the algorithms they were implemented in the language C and a computer experiment was carried out on Sun4 work station.

Altogether the learning by both algorithms $\mathcal{L}$ and $\mathcal{L}_1$ of 1000 randomly generated expressions of star-height from 1 to 3 and of length up to 30 characters were checked. The examples were obtained from the expressions by unfolding the terms with the average factors of

term unfoldings increasing from 3 to 10. In the case of algorithm $\mathcal{L}_1$ a complete coincidence between the original and the synthesized expression was registered in over 70% of cases. In over the third of the remaining cases, the synthesized expression was equivalent to the given. In all the rest of the cases the synthesized expression was roughly equivalent to the original. The results for algorithm $\mathcal{L}$ were approximately the same, but in the majority of cases, the examples for which the success was registered were considerably longer than the respective examples for algorithm $\mathcal{L}_1$.

Thus, in a sense, the learning algorithms were 100% successful. Still these results should be regarded with caution. Strictly speaking, the only fact the experiment has demonstrated is that for expressions and examples randomly generated by the particular method used in the experiment, the algorithms "almost always" work correctly, at least, to within the rough equivalence. To implement a more correctly staged experiment, the methods of generating random "interesting" expressions and examples should be studied and substantiated.

5 Conclusions and Future Research

Although the presented experimental results should be regarded only as preliminary, they suggest that the proposed learning algorithm $\mathcal{L}_1$ may be quite practical for identification of regularities in strings of characters. Possible areas of applications may be program synthesis from sample computations as in [5] or genetics. The first part of the paper can be regarded as a theoretical framework of the presented research into learning from "good" examples. "Good" examples are just natural examples coming from a "good" teacher and, to our conviction, this kind of learning may have even more practical applications as research into how to outsmart a teacher who does his best not to teach while formally remaining a teacher. Interesting, that it is difficult to prove theoretically any satisfactory assertion concerning the generalized algorithm $\mathcal{L}_1$ even as strong as for algorithm $\mathcal{L}$, although statistically $\mathcal{L}_1$ works better than $\mathcal{L}$. This may be a characteristic situation in computational learning theory in general that "natural" learning algorithms are in fact better than it can be proved theoretically.

As a possible direction for future research we would like to note the extention of the given algorithms by adding membership queries for refining the rough equivalence. Seems, that at least the class of regular expressions which excludes terms $(T)^*$ such that there exists A for which $A^n \in T$ for $n > 1$ can be learned exactly from a finite set of long enough c-uniform examples and using membership queries.

6 Acknowledgements

A substantial part of the reported research was carried out while the author was at the CS Department of New Mexico State University where he had a good access to Sun work stations what was essential for carrying out the experimental part of the research. The author is very grateful to the faculty and staff of NMSU and, particularly, to Prof. Juris Reinfelds for supporting the work.

References

[1] D.Angluin. *Equivalence queries and approximate fingerprints.* In proceedings of the 1989 Workshop on computational Learning Theory, Morgan Kaufman, San Mateo, CA, August 1989.

[2] D.Angluin. *A note on the number of queries to identify regular languages.* Information and Computation, 51:76-87, 1981.

[3] D.Angluin. *Learning regular sets from queries and counterexamples.* Information and Computation, 75(2):87-106,1987.

[4] G.Barzdin, J.Barzdin. *Rapid construction of algebraic axioms from samples.* Theoretical Computer Science, 90, 1991, p.199-208

[5] J.M.Barzdin. *Some rules of inductive inference and their use for program synthesis.* In proceedings of IFIP, 1983, North Holland, 333-338.

[6] A.Brazma. *Inductive synthesis of dot expressions.* Lecture Notes in Computer Science, 502, 156-212, 1991.

[7] A.Brazma. *Learning a subclass of regular expressions by recognizing periodic repetitions.* Proceedings of the Fourth Scandinavian Conference on AI. IOS Press, the Netherlands (in print).

[8] W.W.Cohen. *Learning Restricted Classes of Regular Languages Using Loop Induction.* Technical Memorandum, AT& T Bell Laboratories, Dec. 12, 1990.

[9] R.Freivalds, E.Kinber, R.Wiehagen. *Inductive inference from good examples.* Lecture Notes in Artificial Intelligence, 397, 1-18, 1989.

[10] E.M.Gold. *Language identification in the limit.* Inform. contr., 10:447-474, 1967.

[11] E.Kinber. *Learning a class of regular expressions via restricted subset queries*, Lecture Notes in Artificial Intelligence, 642, 232-243, 1992.

[12] L.Pitt. *Inductive Inference, DFAs, and Computational Complexity.* Lecture Notes in Artificial Intelligence, 397:18-44, Springer-Verlag, 1989

[13] N.Tanida, T.Yokomori. *Polynomial-time identification of strictly regular languages in the limit*, IEICE Trans. Inf. & Syst., V E75-D, 1992, 125-132.

[14] L.G.Valiant. *A theory of the learnable.* Comm. Assoc. Comp. Mach., 27(11):1134-1142, 1984.

[15] R.Wiehagen. *From inductive inference to algorithmic learning.* Proc. Third Workshop on Algorithmic Learning Theory, ALT'92, Sawado, 1992, 13-24.